12-2014

# A Secure Communication System for Early Childhood Collaboration System

Hao Kang
*Utah State University*

# A SECURE COMMUNICATION SYSTEM FOR EARLY CHILDHOOD COLLABORATION SYSTEM

by

Hao Kang

A report submitted in partial fulfillment
of the requirements for the degree

of

MASTER OF SCIENCE

in

Computer Science

Approved:

| | |
|---|---|
| _____ | _____ |
| Dr. Stephen W. Clyde<br>Major Professor | Dr. Xiaojun Qi<br>Committee Member |

_____

Dr. Ming Li
Committee Member

UTAH STATE UNIVERSITY
Logan, Utah

Dec, 2014

# ABSTRACT

A Secure Communication System for Early Childhood Collaboration System

by

Hao Kang, Master of Science

Utah State University, Dec, 2014

Major Professor: Dr. Stephen W. Clyde
Department: Computer Science

The *Early Childhood collaboration System* (*ECCS*) is a distributed health data system which provides coordinated, de-identified healthcare information to various types of data consumers. To satisfy this requirement, the ECCS needs a matcher that utilizes *Personal Identifying Information* (*PII*) to coordinate information from a wide range of data sources. Due to the sensitivity of PII, the ECCS also needs to guarantee that only matcher can access PII. This report describes a reusable subsystem, called *Medical Records Secure Messaging* (*MRSecureMessaging*), which utilizes a reliable cryptographic algorithm to encrypt PII and other confidential data so that matcher and data consumers cannot access each other's data. It also contains a customized authentication protocol that allows data sources to verify the intended recipient.

(64 pages)

# ACKNOWLEDGMENTS

I am deeply indebted to my advisor, Dr. Stephen W. Clyde, for his invaluable advice, insightful guidance and constant patience. Without his persistent help, this dissertation would not have been possible.

I would also like to thank my committee members, Dr. Ming Li and Dr. Xiaojun Qi (in alphabetical order), for their support and suggestions.

Many thanks to Vicki Anderson, Cammie Dodds and Genie Hanson (in alphabetical order); their great work and help were indispensable for completing the report.

Finally, I am grateful to my family and friends for their continuous support and encouragement through all my academic pursuits.

Hao Kang

# CONTENTS

# LIST OF TABLES

Table Page

# LIST OF FIGURES

Figure Page

# CHAPTER 1

# INTRODUCTION

A *collaborative person information system* (*CPIS*) is a software system that integrates and coordinates data from a diverse set of data sources in a way that allows various data consumers to have de-duplicated and unified views of all that is known about a person across those data sources. In healthcare, a CPIS deals with patients' *personal identifying information* (*PII*), their health histories, medical conditions, clinical data, lab results, economic data, and much more. This kind of information is extremely valuable for social and medical research, but is also very sensitive data.

A CPIS must secure data carefully to guarantee privacy and confidentiality. Privacy, in the context of healthcare, is the patient's right to determine who see his or her information and for purposed they must use it [24]. Confidentiality is the system's responsibility of ensure that no unauthorized person accesses a person's data and that authorized person only use it for the prescribed purposes [24]. Security is the policies, physical protections, and electronic protections that an organization uses to enforce confidentiality [24]. In essence, privacy of an individual's health information depends on the amount of control and level of confidentiality that a health data system affords, which in turn depends on the security measures that stewards of the health data system implemented.

The *Utah Department of Health* (*UDOH*) contracted with *Utah State University*

(*USU*) to build a distributed health data system, called *Early Childhood Collaboration System* (*ECCS*), that reuses and extends a CPIS, namely the *Public Health Master Patient Index* or *phMPI*. ECCS's high-level requirements include:

- collecting child health care and educational data from 11 distinct data sources (participants) with the possibility of expanding set several times over,
- correlating that data so an individual's records from one source matches his/her records from other sources without duplicates,
- de-identifying that combined data, and then,
- sending those correlated data to a data consumer, which will store them in a data warehouse for later analysis.

Figure 1-1 and Figure1-2 provide a high-level architectural view of ECCS. The orange components represent the data sources; the purple components are the CPIS, which handles the data correlation and transmission; and the green components represent a target data consumer.

As mentioned above, the initial system will have 11 data sources. Table 1-1 lists these sources. Each data source uses a different kind of database manager and different data scheme. The heterogeneities among the data sources are an interesting problem, but beyond the scope of this report.

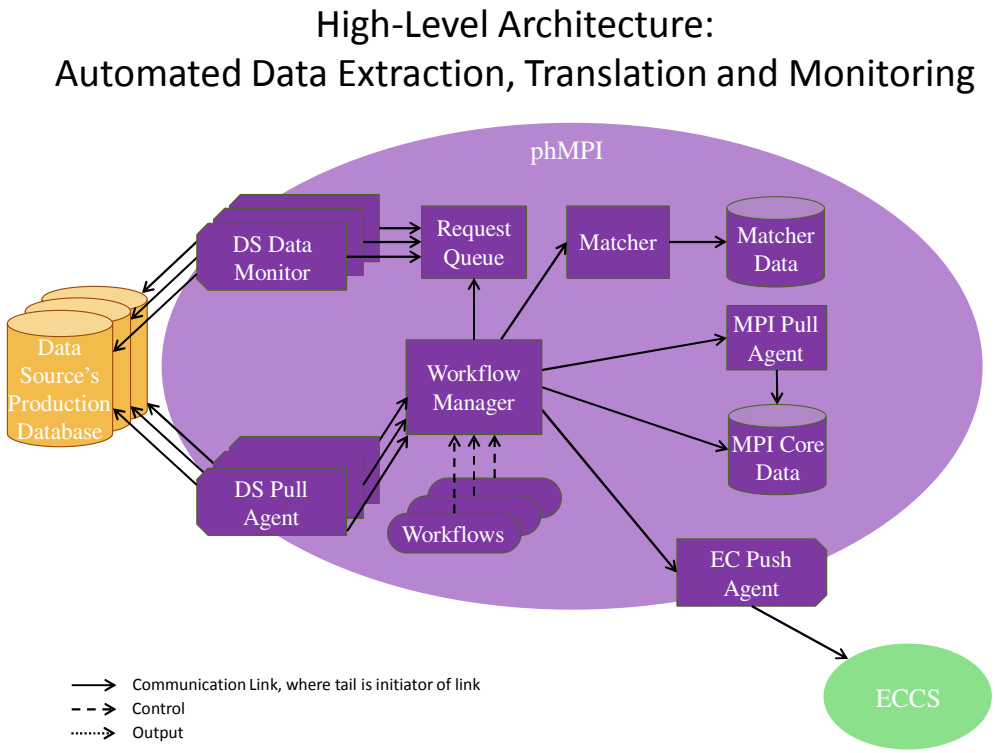


Figure 1-1: phMPI Components

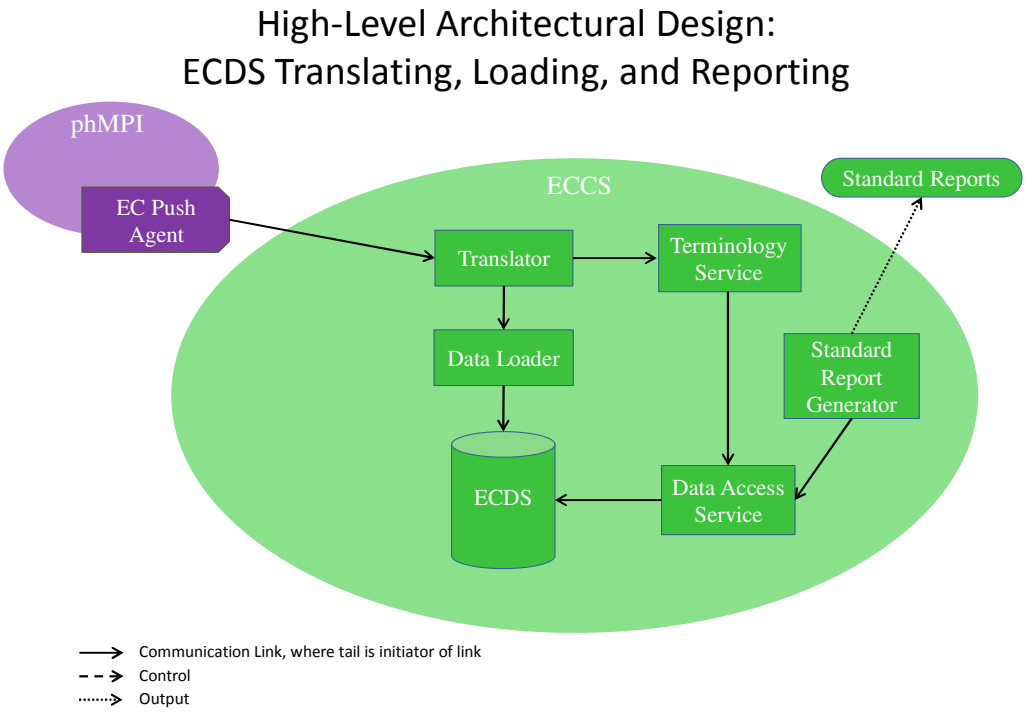


Figure 1-2: ECCS Components

Table 1-1: Data sources that need to feed ECCS (at the time of the report)

| Agency | Software System(s) |
|---|---|
| Early Intervention – Part C | BTOTS |
| Child Care | CC |
| Foster Care | FC |
| Head Start | Multiple Systems |
| Help Me Grow | HMG |
| Early Hearing Detection and Intervention | HiTrack |
| Office of Home Visiting | OHV |
| United Way | UW |
| Utah State Wide Immunization Information System | USISS |
| Office of Vital Records and Statistics | VS |
| Utah State Office of Education | UTREx |

As mentioned above, the CPIS for ECCS is the phMPI, see Figure 1-1. The phMPI is itself a distributed system, designed around a service-oriented architecture and built on top of *Vitruvian* framework. Vitruvian was initially built at USU by a master student to facilitate object distribution and inter-object communications [22], but USU licenses it to a company, namely *Multimedia Data Services Corporation*, for commercialization. More information about Vitruvian framework is provided in Chapter 2 Section 2.3.

The phMPI includes several types of independent services or agents. First, there

are *Data Monitors* for each data source. A *Data Monitor* periodically checks its data source for new or changed records and sends processing requests to a central *Request Queue* when it detects such events. A *Workflow Manager* grabs the processing requests out of the *Request Queue* and executes a specific *Workflow* for each one, depending on the type of request, the data source, and what kinds of data a consumer needs to know about that new or changed record. The *Workflow* may involve any number of actions, including matching person identities with those from other data sources, querying additional person data from the data source on which the event was detected, extracting person data from other data sources, merging all of the person's data in a combined person snapshot, and publishing that snapshot to one or more data consumers.

Out of necessity, matching operations require PII to find person identities from other data sources that represent the same person as the one for which a new or changed event was detected. In other words, the phMPI needs to know certain amount confidential information. However, it doesn't need to have access to complete health histories, medical conditions, clinic data, lab results, or other personal details.

To handling the querying of additional person data, the phMPI includes *Pull Agents* that can retrieve, in real time, a snapshot of given person's data, specific its data source. There is a custom-configured Pull Agent for every data source.

The publishing of combined person snapshots is handled through *Push Agents*, one configured for each possible data consumer. For ECCS, the data consumer of

interest is the *EC Data Analysis System* shown in Figure 1-2. When publishing, the *Workflow Manager* first sends a combined person snapshots to the *EC Push Agent*, which in turn sends them to the *Translator* inside the *EC Data Analysis System* (*ECDS*). The *Translator* transforms the incoming data into its own format and converts any embedded terminology into its own lexicon using a *Terminology Service*, and sends the resulting data onto the *Data Loader*, which then stores it in the *EC Data Warehouse* (*ECDW*). A *Report Generator* accesses that data through a *Data Access Service* to create interesting reports that highlight possible trends child health or healthcare.

For the ECCS to guarantee privacy and confidentiality, it has to satisfy the following requirements.

1. The phMPI should only have access to the minimal amount of PII that is needed to accurately match person identities. In other word, the phMPI should not have access to health histories, medical conditions, clinic data, lab result, economic data, or any other details not specifically need for record matching.
2. When a Pull Agent extracts data intended for only the EC Data Analysis System (as requested by a *Workflow*), ECCS must ensure that only the EC Data Analysis System can access that data. Similarly, if a Pull Agent extracts data from other data sources, then ECCS must guarantee that only those intended data consumers can access the data.

3. ECCS must allow individuals to opt out of sharing any type of data with any of the possible data consumers.

This report describes the development of such subsystem named the *Medical Records Secure Messaging* (*MRSecureMessaging*) that addresses the first two requirements. The thirds is handled by another subsystem called *Consent Management System* [23]. MRSecureMessaging includes a customized authentication protocol that allows data sources to verify the intended recipient. It also utilizes a reliable cryptographic algorithm to encrypt confidential data that phMPI and other are not allowed to view. Different recipients' data would be encrypted by different keys so they cannot see each other's data. These two features work together to satisfy the first two requirements.

Figure 1-3 provides a high-level architectural view of the MRSecureMessaging. The red components represent the MRSecureMessaging's encryptors and decryptors. When a Pull Agent tries to grab data, data source first lets the *Encryptor* encrypt it. The Encryptor uses asymmetric cryptography to handle this (more information about asymmetric cryptography is provided in Chapter 2 Section 2.2). As mentioned above, it encrypts PII with phMPI's key and other confidential data with ECCS's key. In this way, when the data arrives at phMPI, the Matcher can only decrypt PII with its own key. It then uses this information to finish correlating. Matcher doesn't have access to other data since it doesn't have the corresponding secret key to decrypt it. When correlating and merging process is done, the Push Agent sends a combined person

snapshot encrypted with ECCS' key to the Translator. Translator decrypts incoming data with its own key and then transforms data into its own format. Finally, the Data Loader sends transformed data to the ECDS.

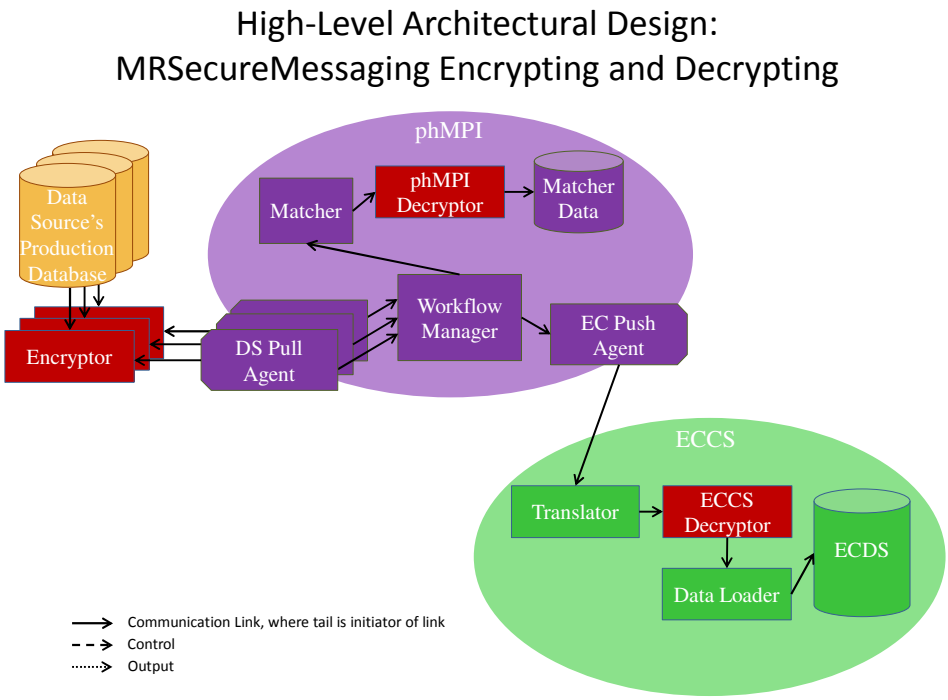


Figure 1-3: Encryptors and Decryptors in the phMPI and ECCS

So far we have a clear understanding of the ECCS and the MRSecureMessaging's requirements. Chapter 2 provides sufficient background knowledge related with them. Then Chapter 3 introduces the theoretical design of MRSecureMessaging. Chapter 4 covers the system analysis of MRSecureMessaging. With the information from these two chapters, I illustrate the architectural design in Chapter 5. Chapter 6 discusses technique details of the implementation and Chapter 7 mainly focuses on the testing process. Finally, in Chapter 8, I summarize the

contribution of the MRSecureMessaging and possible future work.

# CHAPTER 2

# BACKGROUND

The ECCS is dealing with healthcare data, so it is essential to understand the confidentiality of healthcare data to appreciate the contribution of this paper. Section 2.1 provides some high-level background in area. As a subsystem of both the phMPI and ECCS, the MRSecureMessaging encrypts PII and other confidential data with public key cryptographic algorithm. It also contains a customized public key authentication protocol to ensure secure communication. Section 2.2 first explains what a cryptographic algorithm is. It then introduces the concept of authentication protocol and lists three main attacks to authentication protocol. Both phMPI and the MRSecureMessaging are built on top of the Vitruvian framework; having some knowledge about it is necessary. Section 2.3 gives a high-level overview of the Vitruvian framework.

## 2.1 Confidentiality of healthcare data

Person's healthcare data must keep confidential and be handled carefully for the following reasons:

First of all, healthcare data is person's privacy. Person's healthcare data contains PII, such as names, gender, birth date, parents, siblings, children, race, ethnicity, residential location, medical identifiers, and contact information. In general,

PII is any data that can be used by itself or in conjunction with other information to identify an individual. An organization should only request PII if the PII is absolutely necessary [25]. The Likelihood of harm caused by a breach involving PII is greatly reduced if an organization minimizes the amount of PII it uses, collects and stores [25].

Some cases, a person's health history data or medical conditions could also be used to determine identity; but these kinds of information are not typically used by records matchers, so we put them in the category of confidential non-PII data. Besides a health history and medical conditions, a person's non-PII data may include all kinds of details and very sensitive information, such as medications, clinical data, lab results, and economic or financial data. At an individual level, this data is extremely sensitive and must be keep confidential.

Organizations can de-identify records by removing enough PII such that the remaining information does not identify an individual and there is no reasonable basis to believe that the information can be used to identify an individual [25].
De-identified records can be used when full records are not necessary [25]. In ECCS, a data source sends de-identified snapshots of these kinds of information to the ECDS in a way that guarantees that others, and even the phMPI, cannot access them. Then, the ECDS only allows these data to be reported on in aggregate so no single individual data can be view directly.

Secondly, any data requestor must strictly abide by the data-sharing agreement

its steward agency has with the data source. A data-sharing agreement is legal agreement that defines the terms of data access [26]. It may include:

- Intent and scope of data sharing
- Potential benefits (including projected efficiencies) and risks of sharing, benefits and risks of not sharing , and methods to monitor these benefits and risks
- Methods that will be used to share data and roles and responsibilities of staff involved
- Minimum data elements needed to achieve the objective(s), including need for PII
- Steps that will be taken to ensure the confidentiality and security of shared data
- Provisions for physical and electronic security
- How shared data will be used, analyzed, published, released, and retained/destroyed
- Confidentiality agreements
- Knowledge and training requirements including annual training for staff who have access to PII and non-PII data

ECCS has data-sharing agreement with its 11 data sources to share all kinds of patient data, but phMPI only has data-sharing agreements that grant it access to PII. As mentioned in Chapter 1, the ECCS collects data from a wide range of data sources

and reuses phMPI's matcher to correlate that data. When multiple medical records with the same patient name come to the matcher, the phMPI uses PII to determine whether these records belong to the same person. So, the systems must ensure that the phMPI cannot access non-PII data as it flows from the data sources through the phMPI to ECCS.

Finally, federal laws such as *Health Insurance Portability and Accountability Act* and *Confidentiality of Alcohol and Drug Abuse Patient Records*, limit the use of healthcare data. But this is beyond the scope of this report.

## 2.2 Introduction to Secure Communications

Secure communications are needed when two principals exchange messages and want to guarantee that a third party cannot access the message content, modify that content, or reproduce/resend it [20]. One way to realize secure communication is hiding the content or nature of messages, namely encryption. In addition, secure communications also requires each of the principals to authenticate their identity so other principal has some assurance that it communicates with the right entity. An authentication protocol can address this requirement.

### 2.2.1 Overview of Encryption and cryptographic algorithm

Encryption is the process of encoding messages or information in such a way that only authorized parties can read it [28]. A concept closely related with encryption is cryptographic algorithm. In an encryption scheme, the message or information,

referred to as *plaintext*, is encrypted using a cryptographic algorithm, generating *ciphertext* that can only be read if decrypted [27].

Cryptographic Algorithms can roughly be divided into three different types: Symmetric Key Cryptography, Public Key Cryptography and One-way Hash Algorithm [1].

Symmetric Key Cryptography is a class of algorithms that use the same cryptographic keys for both encryption and decryption [2]. To ensure security of communication, keys are kept secret between the communicating principals [1]. However the requirement that both principals have access to the key is one of the main drawbacks of symmetric key cryptography [2]. Carelessness of either principal will make key to compromise. Modern-day symmetric key algorithms are principally *block ciphers* or *stream ciphers* [1]. A block cipher will encrypt a block of (typically 64 or 128) plaintext bits at a time. The best known block cipher is the Data Encryption Standard [7]. In contrast, stream ciphers encrypt one bit of plaintext at a time.

Public Key Cryptography, also known as asymmetric cryptography, is a class of algorithms that require two separate keys, one of which is secret (or private) and the other is public [3]. The public key is made publicly available but the principal would never reveal its private key. Although different, the two parts of this key pair are mathematically linked [3].Unlike symmetric key cryptography, in public key cryptography there is no shared secret between communicating parties [1]. Usually

(but not always) public key is used for encrypt information and secret key is used for decrypt. In 1976, Whitfield Diffie and Martin Hellman published the first paper on this topic [4]. The most widely known public key algorithm was developed by Rivest, Shamir and Adleman [5] and is universally referred to as RSA. Comparing to symmetric key cryptography, public key cryptography's encryption speed is considered slower.

One-way Hash Algorithm maps an input to a hash value of specific length. This hash value is often referred to as a *digest*. The mapping of inputs to digests is one-way; so it is practically impossible to recreate an input from its hash value alone. It is also infeasible to find two inputs with the same hash values. Hence, the recipient of a message could use a hash algorithm to yield its own digest, and then compare this digest with the received digest to check whether the received message was modified. This type of algorithms is intended for use in conjunction with cryptography to provide signatures [1].

### 2.2.2 Overview of Authentication Protocols

Before introducing authentication protocol, we need to understand another concept first, namely that of a *cryptographic protocol*. A cryptographic protocol is an abstract or concrete protocol that performs a security-related function and applies cryptographic algorithm [29]. Cryptographic protocols are widely used for secure application-level data transport. An authentication protocol is a type of cryptographic protocol with the purpose of authenticating entities wishing to communicate securely

[30].

According to the cryptographic algorithm taken, authentication protocols could be categorized into two main types: symmetric key protocol and public key protocol. Further distinctions are made on whether they use *trusted third parties* (*TTP*) or not. Notice that authentication protocols could also be categorized by many other different ways, but we will not discuss them in this report.

The first type is *Symmetric Key Protocol without TTP*. This type of protocol operates purely between two communicating principals that wish to achieve some mode of authentication [1]. Communicating principals need to share a key before sending messages to each other. Keys may be generated by a trusted organization and send to each principal individually. This requirement restricts them from changing the shared key frequently. Once the key is compromised, a malicious principal could keep eavesdropping on the network until they change it.

To improve this, the second type, *Symmetric Key Protocol with TTP*, was introduced. This type of protocol uses TTP to carry out some agreed function [1]. Usually third parties are trusted for key generation and distribution, but they may be trusted for activities other than that. With TTP, principals could change shared keys every time they plan to communicate so security is improved.

The third type is *Public Key Protocol*. This type of protocol also needs TTP to distribute principals' public key. When a principal wants to talk to another principal, it needs to get the other one's public key from TTP first. Then use this public key to

encrypt messages. On the other side, the recipient is able to decrypt messages with its own secret key.

As mentioned above, public key cryptographic algorithms are usually slower than symmetric key cryptographic algorithms. This also restricts the efficiency of corresponding protocols. To achieve a better efficiency without compromising security too much, *Hybrid Protocol* was introduced. This type of protocol uses both public and symmetric key cryptography. The trick is exchanging symmetric encryption keys using public key cryptography.

Protocols can fail in many different ways. The following paragraphs discuss three main types of attack related with secure communication.

Freshness attacks. A freshness attack occurs when a message (or message component) from a previous run of a protocol is recorded by an intruder and replayed as a message component in the current run of the protocol [1]. This type of attack could be fixed by the use of timestamps. Every time when a principal receives a message, it first checks whether the timestamp is within an allowable clock skew or not. If yes, it responds that message. If no, it discards that message and logs an error.

Man-in-the-middle. This type of attacks is a form of active eavesdropping in which the attacker makes independent connections with the victims and relays messages between them, making them believe that they are talking directly to each other over a private connection, when in fact the entire conversation is controlled by the attacker [21]. A man-in-the-middle attack can succeed only when the attacker can

impersonate each endpoint to the satisfaction of the other. This enlightens us that to prevent this attack, we need to find a way to stop impersonating. Chapter 3 discusses this in details.

Guessing attacks. This type of attacks happens when the intruder grains useful information from communicating messages and then uses this information to verify his guess. This attack can be very dangerous when the intruder has powerful computational resource and is able to verify his guess off-line. To avoid this attack, the protocol designers need to ensure that data available to intruder is sufficiently unpredictable [8].

## 2.3 Overview of the Vitruvian Framework

Technologies and frameworks that support communication, distribution, and replication for distributed systems are not new. However they all suffer from one problem which is exposing too much communication and distribution details to developers. Vitruvian is a service-oriented distribution framework that tries to minimize developers' effort to distribute objects, without compromising functionality, extensibility, good modularization, performance, or maintainability.

Vitruvian mainly handles two challenges:

1. Designing new distributed systems from the ground up
2. Accommodating new requirements for distribution in mid development stream.

MRSecureMessaging is built on top of Vitruvian, which help the developer

achieve a reasonable degree of access, location, and communication transparencies. Transparency is defined as the concealment from the user and the application programmer of the separation of components in a distributed system, so that the system is perceived as a whole rather than as a collection of independent components [16]. The implications of transparency are a major influence on the design of the system software [16]. Specifically, *access transparency* enables local and remote resources to be accessed using identical operations [16]. *Location transparency* enables resources to be accessed without knowledge of their physical or network location [16]. *Communication transparency* enables different parties to communicate without knowing the low-level details. Vitruvian distributes objects using dynamically generated proxies which are specializations of the application classes. Vitruvian also provides various synchronization patterns for proxies. Chapter 6 Section 6.1 provides this information in details.

# CHAPTER 3

# THEORETICAL DESIGN

The MRSecureMessaging adapts and extends Needham-Schroeder public key protocol for authentication and encryption. It also utilizes the public key cryptography to enforce the data access constraint. Considering sensitive data is distributed and exchanged, the protocol must be stable. Unfortunately, some protocols are found to be flawed many years after their publications. Actually Needham-Schroeder protocol itself is an example. Lowe found a flaw (will be discussed in Section 3.3) 17 years after it was first published. However after many corrections and refinements, it is now considered a reliable protocol.

Secondly, public key cryptography algorithms can provide better secrecy and access control. It could cause serious consequences once children's healthcare information is compromised. The MRSecureMessaging and the ECCS must guarantee data's privacy, confidentiality and security. More importantly, public key algorithms can easily handle data access issues. Different data is encrypted with different viewer's public keys; only corresponding private keys could decrypt it. Getting others' private keys is very hard since private keys are never distributed through network and always stored safely by their owner (Symmetric keys are distributed through network and therefore have greater chance to compromise). In sum, although public key

algorithms are generally slower than symmetric key algorithms, considering the problems mentioned above, it is still preferred. Also considering that computing power is increasing rapidly, the speed of encryption and decryption is acceptable.

Finally, using public key algorithm simplifies the communication flows. Suppose a symmetric key algorithm is used, then data sources need to communicate with phMPI and ECDS individually to distribute their symmetric keys (generated by a key server), which in turns increases the MRSecureMessaging's complexity. In contrast, when public key algorithm is used, data sources only need to communicate with phMPI.

Section 3.1 introduces the notational conventions used in this report; Section 3.2 reviews the Needham-Schroeder public key protocol; Section 3.3 lists a few modifications to the original protocol and explains the reasons; Section 3.4 gives a summary of the MRSecureMessaging's theoretical design.

## 3.1 Notational Conventions

In this report, I use the notation $\{M\}_{PK(A)}$ to denote the result of encrypting message plaintext $M$ with public key of principal $A$. $PK$ stands for public key and $SK$ stands for secure key. In general, capitals denote principals, such as $A$, $B$, $S$ (for a server) and $I$ (for an Intruder). $A$ sends a message $M_1$ to $B$ would be denoted like this:

$$A \rightarrow B: M_1$$

$N_a$ denotes a random number generated by principal $A$. Such numbers are intended to be used only once for the purposes of the current run of the protocol and

are generally termed *nonces* [1]. $C_a$ denotes a confounder generated by principal *A*. Section 3.3 will explain this concept in details.

A message may have several components; some will be plaintext and some will be encrypted. Message components will be separated by commas. *Unified Modeling Language* (*UML*) sequence diagram is used to describe protocols' message flows.

## 3.2 Introduction to Needham-Schroeder Public Key Protocol

Needham and Schroeder proposed this protocol in 1978 [6]. Figure 3-1 illustrates the message flows of the protocol.

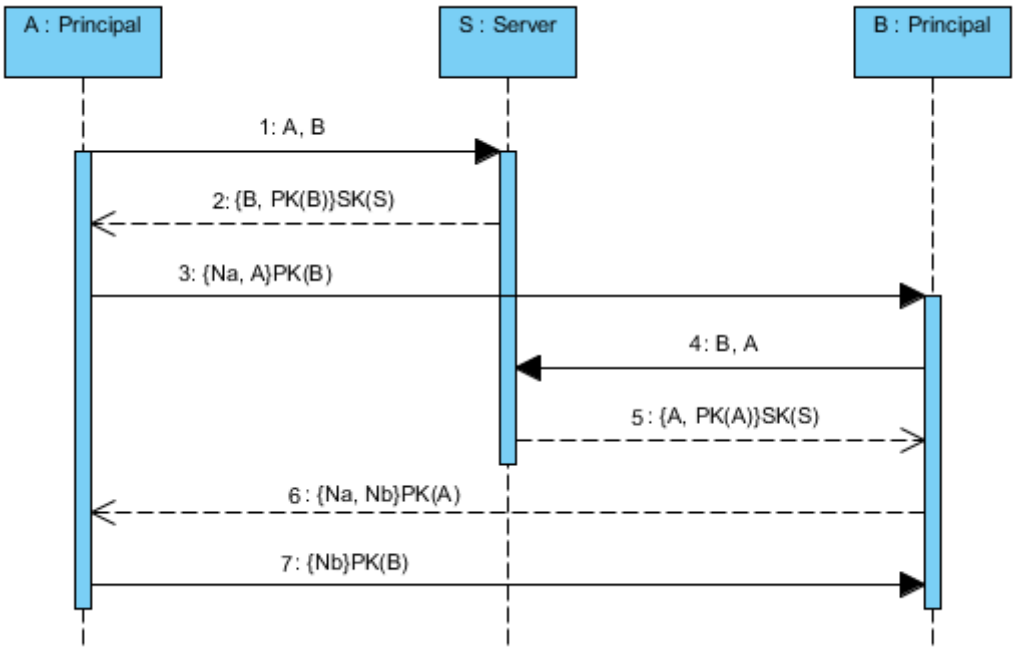


Figure 3-1: Needham-Schroeder public key protocol

In this protocol, server *S* stores all principals' public keys and distributes them when request is received. Principal *A* first uses Messages 1, 2 to obtain principal *B*'s

pubic key. *A* then encrypts a nonce and its own identifier with by *B*'s public key. *B* receives Message 3 and decrypts it to obtain *A*'s nonce. Similarly, *B* uses Message 4, 5 to obtain *A*'s public key. After that, *B* forms its own nonce and encrypts both nonces with *A*'s public key as Message 6. *A* decrypts Message 6 and knows that *B* is operational and has responded. The reason is that *B* is the only one who has necessary information to create Message 6. *A* then encrypts $N_b$ using *B*'s public key and sends back to *B*. *B* then decrypts and checks if Message 7 contains its nonce. If the answer is yes, *B* concludes that *A* is operational and indeed initiated the protocol.

However, Gavin Lowe showed that this protocol flawed relative to man-in-the-middle attacks [9]. Other than that, it also needs some modifications to fulfill security requirements in the ECCS. These will be discussed in the next section.

## 3.3 Customized Authentication Protocol for the MRSecureMessaging

This section covers several modifications to Needham-Schroeder public key protocol.

### 3.3.1 Adding timestamps into messages

The purpose of timestamp is to make ensure the "freshness" of a message. Each message should contain a timestamp in it. Then, if a message is not received within an allowable clock skew, the recipient can discard it and log a failure. Freshness attacks are therefore avoided.

### 3.3.2 Fixing a man-in-the-middle attack

As mentioned above, Gavin Lowe has showed that to Needham-Schroeder public key protocol is susceptible to man-in-the-middle attack. Figure 3-2 illustrates this kind of attack, where *A* tries to communicate with *B*, but *I* intrudes on that communication. *A* sends the nonce $N_a$ encrypted with *I*'s key. *I* could impersonate *A* to start another session with *B*. *B* replies with a new nonce $N_b$ encrypted with *A*'s key. Although *I* could not decrypt this message, he could simply replay this message to *A*. Finally *A* decrypts this message and return $N_b$ to *I*, *I* decryptes the message and return it to *B*. *B* believes that he has correctly carried out a run of protocol with *A*.

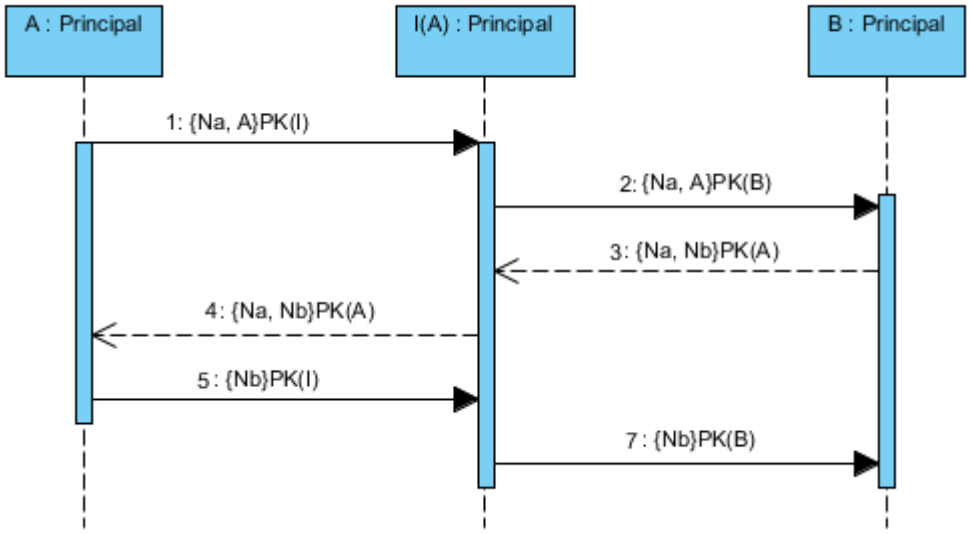


Figure 3-2: A man-in-the-middle attack to Needham-Schroeder public key protocol

To prevent this attack, Lowe suggested including the identity of the responder within the encrypted part of Message 3 (the third message in Figure 3-2, not the third message of the original protocol). In this way, intruder *I* cannot successfully replay it since *A* is expecting a message containing *I*'s identity. The new Message 6 of

Needham-Schroeder protocol looks like this:

$$B \rightarrow A: \{N_a, N_b, B\}_{PK(A)}$$

3.3.3 Adding confounders into messages

A confounder is distinct from a nonce in that it has no purpose other than to confound guessing attack. The value of a confounder may be ignored by the recipient of the message in which it appears [9]. With the help of confounder, it is very difficult for the intruder to verify his guessing so that guessing attacks could be avoided.

Suppose there is an intruder in the system. He may record Message 6 and try to guess the secret key of $A$ to decrypt this message. Then he can get $N_a'$, $N_b'$ and $B'$. If $B' = B$, this guessing might be correct, he could generate two messages: $\{N_a', A\}_{PK(B)}$ and $\{N_b'\}_{PK(B)}$. By comparing these two messages with Message 3 and Message 7, the intruder could verify his guessing (see Table 3-1).

This is a dangerous attack since the process of guessing can be done offline. The intruder may have very powerful computing recourse to make guessing and verifying. To avoid from this attack, confounders are added to the messages to stop the intruder from verifying his guessing (see the last column of Table 3-1). " $\oplus$ " stands for XOR. Suppose an intruder decrypts new Message 6 with his guessing key, he could get $(N_a \oplus C_{a1})'$ and $N_b'$, but there is no way for him to verify his guessing due to the changings of Message 3 and Message 7. Therefore, guessing attacks are avoided.

Table 3-1: Fixed messages in Needham-Schroeder protocol

| Message # | Original messages | Intruder's messages | Fixed messages |
|---|---|---|---|
| Message 3: | $\{N_a, A\}_{PK(B)}$ | $\{N_a', A\}_{PK(B)}$ | $\{N_a, A, C_{a1}, C_{a2}, T_{a1}\}_{PK(B)}$ |
| Message 6: | $\{N_a, N_b, B\}_{PK(A)}$ | N/A | $\{N_a \oplus C_{a1}, N_b, B, T_{b1}\}_{PK(A)}$ |
| Message 7: | $\{N_b\}_{PK(B)}$ | $\{N_b'\}_{PK(B)}$ | $\{N_b \oplus C_{a2}, C_{a3}, T_{a2}\}_{PK(B)}$ |

Figure 3-3 depicts the customized Needham-Schroeder public key protocol (notice that confounders have been added into messages).

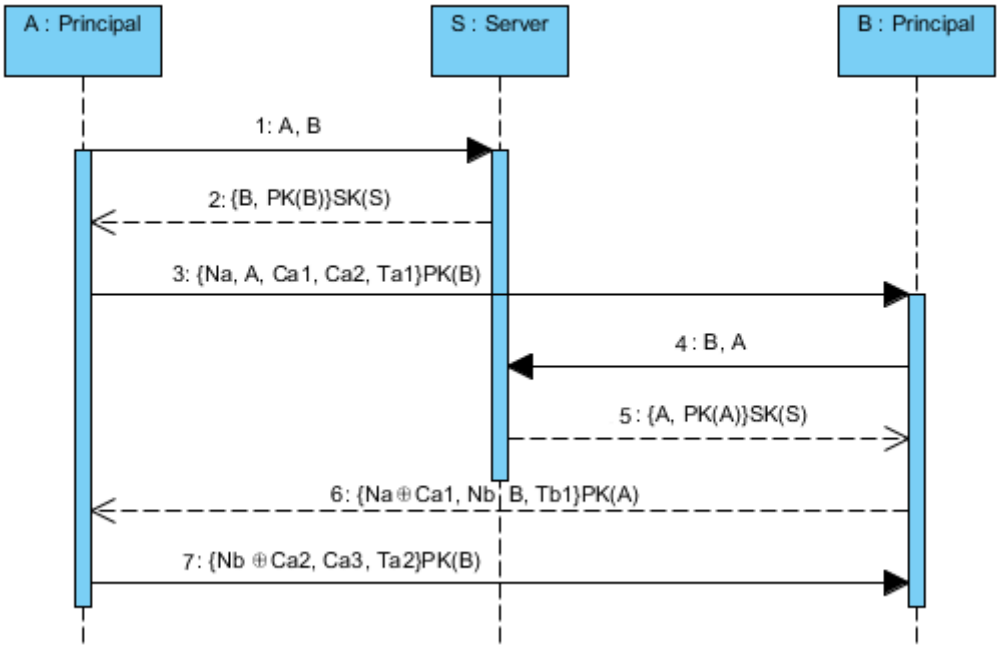


Figure 3-3: Customized Needham-Schroeder public key protocol

## 3.4 Summary

Figure 3-4 describes a high level summary of the MRSecureMessaging (coordinating process is not included in this diagram since that is not part of the

MRSecureMessaging). The working flow is as follow:

1. *A* requests *B* and *C*'s public key separately.
2. KeyServer *S* replies *A* with corresponding public keys.
3. *A* authenticates *B* with customized Needham-Schroeder public key protocol.
4. *A* encrypts *Data1* and *Data2* with *B*'s public key and *C*'s public key.
5. *A* sends data to *B*.
6. *B* authenticates *C* with customized Needham-Schroeder public key protocol.
7. *B* sends *Data2* to *C*.

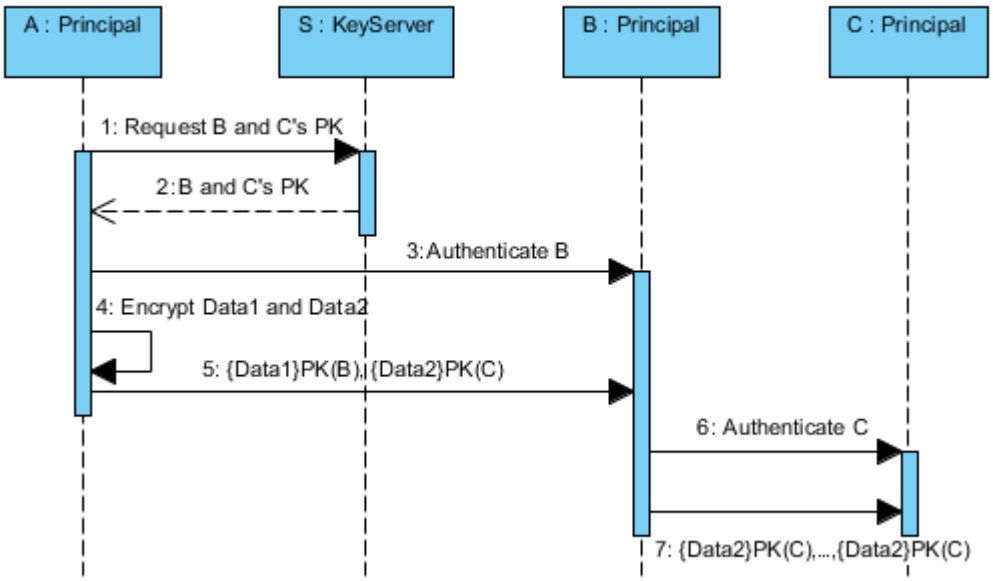


Figure 3-4: A high level summary of the MRSecureMessaging

# CHAPTER 4

# SYSTEM ANALYSIS

System analysis is "the process of studying a procedure or business in order to identify its goals and purposes and create systems and procedures that will achieve them in an efficient way" [10]. Specifically, in software engineering, the purpose of system analysis is to understand and document the essential characteristics of the system being studied. However, these types of documents could be highly-abstracted and hard to understand. I used UML diagrams to visualize the system analysis and again here to communicate the analysis to the readers. The UML is a general-purpose modeling language which is designed to provide a standard way to visualize the design of a system [11].

The use-case diagrams in Section 4.1 provide a high-level overview of actor and its goals. The class diagrams in Section 4.2 summarize the key objects in the system and their relationships to each other from an analysis perspective. The functional requirements listed in Section 4.3 details these goals. Non-functional requirements listed in Section 4.4 specify the criteria that can be used to judge the operation of the system.

## 4.1 Actor Goals

The actor is any external entity that has an interest in interacting with the

system. Often, it is a human user of the system, but it can also be another system or some kind of hardware device that needs to interact with the system [11]. Actor's goals are captured by use-case diagrams. A use-case diagram shows actors, use cases and their relationships [11].

Three main actors closely relate with the MRSecureMessaging. Figure 4-1, 4-2 and 4-3 describe their primary goals.

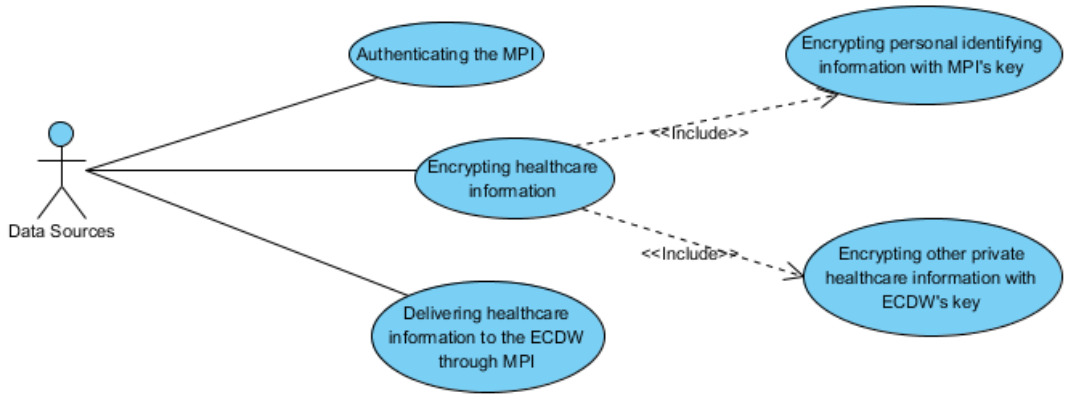


Figure 4-1: Data sources' primary goals

Data sources have three primary goals. First, they need to authenticate MPI to make sure healthcare data is sent to the "right" place. Second, they should be able to encrypt data. This goal includes two sub goals. In short, data sources needs to encrypt different data with different recipients' public keys. Third, data sources need to deliver healthcare data to the ECDW through MPI since the data needs to be coordinated and de-identified in MPI.

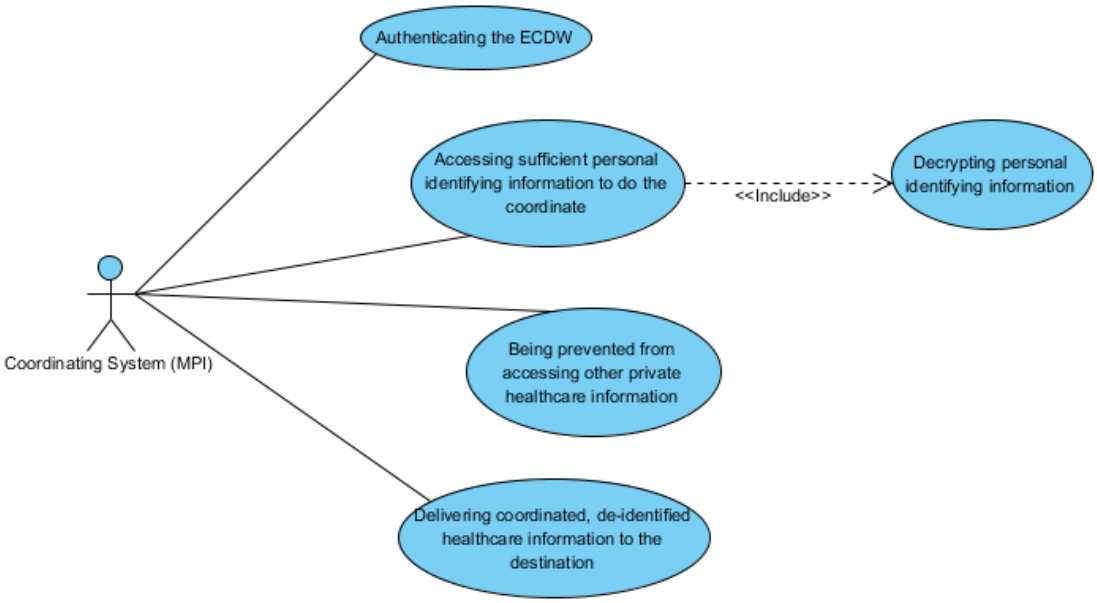


Figure 4-2: MPI's primary goals

MPI has four primary goals. First, similar to data sources, it needs to authenticate the ECDW. Second, it should have access to PII; this one includes decrypting received data with its own private key. Third, it must not have access to other confidential healthcare data. Fourth, it needs to deliver coordinated, de-identified data to the ECDW.

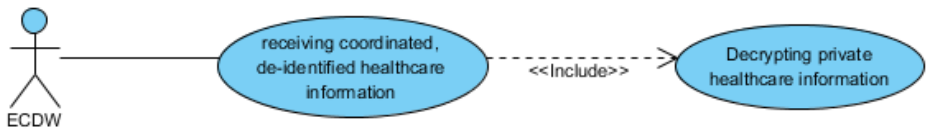


Figure 4-3: ECDW's primary goals

The ECDW only has one goal, namely the receiving of coordinated,

de-identified data from MPI. This goal includes one sub goal that it needs to be able to decrypt received data with its own private key.

## 4.2 Object-oriented Structural Analysis

Structural analysis could give developers a better understanding of a system's components and structure. UML class diagrams are used to illustrate such analysis. A class diagram describes the static view of a system in terms of classes and relationships among the classes [11]. Class diagrams are not just for visualizing and documenting structure models, but also for constructing an executable system. An object-oriented programming language can directly implement a class, making the class diagram one of the core diagrams for generating code and making UML executable [11].

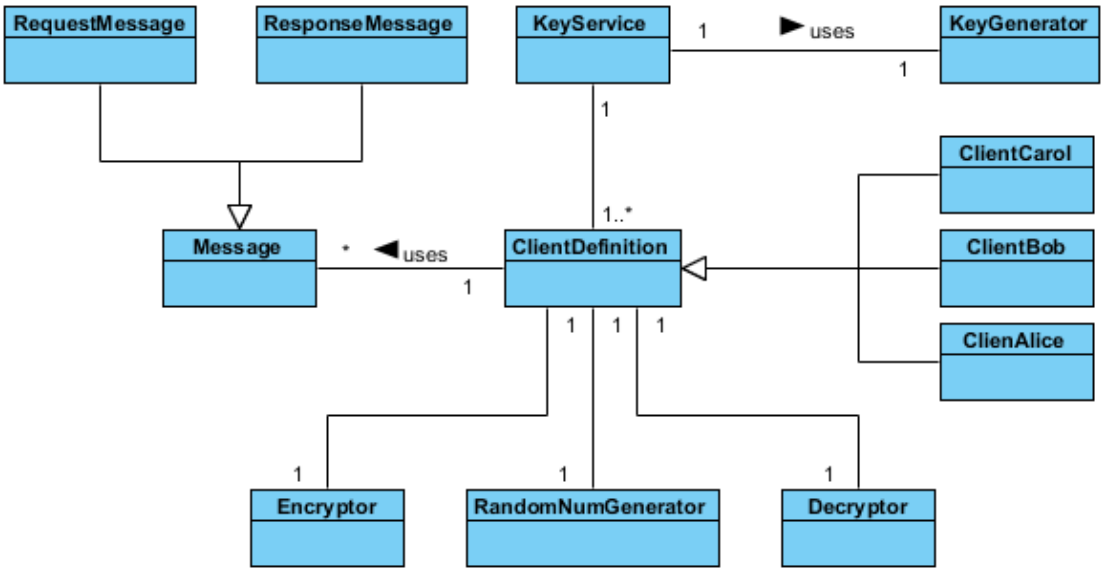


Figure 4-4: Class diagram of the MRSecureMessaging

Figure 4-4 shows the class diagram of the MRSecureMessaging. *KeyService*

stores keys generated by *KeyGenerator* and distributes them when key requests are received. Clients use messages to request keys from *KeyService*; they also use messages to communicate with each other. Each client also has relations with three other classes: *Encryptor*, *Decryptor* and *RandomNumGenerator*. The first two's jobs are handling cryptography and the last is used to generate nonces and confounders.

The following subsections describe the purpose of these classes and design considerations.

### 4.2.1 Key-related classes

The *KeyGenerator* class is responsible for generating both public key and secret key. *KeyService* stores clients' public keys and clients store their own secret keys. As the trusted third party, *KeyService* also distributes public keys when requests are received from clients. Each class has a clear and independent functionality so that flexibility and reusability are achieved. *KeyGenerator* could be used in other projects. *KeyService* could also easily substitute current *KeyGenerator* with another one.

### 4.2.2 Client-related classes

*ClientDefinition* defines the common properties of all clients. This design avoids redundant code and reflects a good practice of encapsulation. Various types of concrete client classes could inherit from it. Currently there are three: *ClientAlice*, *ClientBob* and *ClientCarol*, which represent the three different parties mentioned in previous chapters. More clients could be added in the future. Clients also need to

handle cryptography, nonces and confounder issues. These requirements are fulfilled by three independent classes: *Encryptor*, *Decryptor* and *RandomNumGenerator* so that they can be reused.

### 4.2.3 Message-related classes

Messages could be categorized into two types: *RequestMessage* and *ResponseMessage*. These two classes plus *Message* class capture common properties of messages. A few concrete message classes extend these two classes. This design is very flexible, since new message class can be added easily by simply adding additional specializations. Figure 4-5 describes different message classes and their relationships.

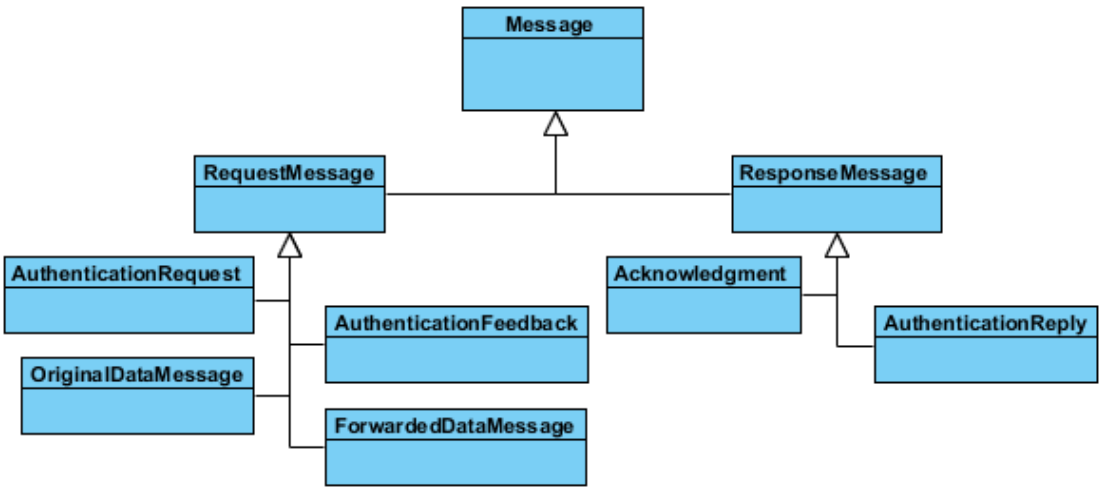


Figure 4-5: Class diagram of messages

## 4.3 Functional Requirements

In software engineering, functional requirements capture the intended behavior of the system. This behavior may be expressed as calculations, technical details, data

manipulation or other specific functionality the system is required to perform [12]. The detailed functional requirements for the MRSecureMessaging are as follows:

4.3.1 Data Sources Requirements

1. The MRSecureMessaging should allow data sources to authenticate the MPI.
2. The MRSecureMessaging should allow data sources to encrypt healthcare information.
    2.1 The MRSecureMessaging should allow data sources to encrypt PII with MPI's public key.
    2.2 The MRSecureMessaging should allow data sources to encrypt other private healthcare information with ECDW's public key.
3. The MRSecureMessaging should allow data sources to deliver healthcare information to the ECDW through MPI.

4.3.2 MPI Requirements

1. The MRSecureMessaging should allow MPI to authenticate the ECDW.
2. The MRSecureMessaging should allow MPI to access sufficient PII to coordinate data.
    2.1 The MRSecureMessaging should allow MPI to decrypt PII with its own private key.
3. The MRSecureMessaging must prevent MPI from accessing other private

healthcare information.

4. The MRSecureMessaging should allow MPI to deliver coordinated, de-identified healthcare information to the ECDW.

#### 4.3.3 ECDW Requirement

1. The MRSecureMessaging should allow ECDW to receive coordinated, de-identified healthcare information.
    1.1 The MRSecureMessaging should allow ECDW to decrypt received information with its own private key.

### 4.4 Non-functional Requirements

A non-functional requirement defines how a system is supposed to be, in other words, it is a requirement that specifies criteria that can be used to judge the operation of a system, rather than specific behaviors [13]. Followings are MRSecureMessaging's non-functional requirements:

1. Operating System
    1.1 The MRSecureMessaging should be compatible with the Microsoft Windows operating system.
2. Languages and Platform
    2.1 The programming language should be C# (C-Sharp).
    2.2 UML should be used to document the system analysis, specification and architectural design

2.3 The system should use Vitruvian for distribution.

3. Quality Control

3.1 The system should have comprehensive unit tests for each class.

3.2 Integration testing is essential to ensure functionality, reliability and performance of the MRSecureMessaging.

4. Documentation

4.1 Design documents and a report about the system should be given to assist the end users in understanding the design and functionalities of the MRSecureMessaging.

5. Logging

5.1 The system should use Vitruvian framework to support logging.

# CHAPTER 5

# ARCHITECTURAL DESIGN

Software architecture is the process of defining a structured solution that meets all of the technical and operational requirements, while optimizing common quality attributes such as performance, security and manageability. It involves a series of decisions based on a wide range of factors and each of these decisions can have considerable impact on the quality, performance, maintainability and overall success of the application [14]. A good architectural design should have high cohesion and loose coupling so that the application can be easily understood, maintained, reused and tested.

The MRSecureMessaging's design needs to address two concerns. The first one is flexibility. The MRSecureMessaging's classes should be designed with good encapsulation so that if one class is changed later, the others won't be affected. In this way, future update will be greatly facilitated. The second concern is reusability. This includes two aspects: first, the MRSecureMessaging's classes should have very clear functionality so that they could be reused in other projects; second, the MRSecureMessaging should also try to reuse code from other projects, for instance, code from Vitruvian framework. This could help the programmer to focus on solving specific problems and keep them away from handling bottom layer issues. Section 5.1

covers these two concerns by illustrating the packages and reusable components of the MRSecureMessaging. Section 5.2, Section 5.3 and Section 5.4 explain the design decisions and dependencies of individual packages separately.

## 5.1 MRSecureMessaging System Design

The Vitruvian Framework provides a service-oriented architecture that distributes objects using dynamically generated proxies that are specializations of the application classes [15]. It also facilitates serialization and logging.

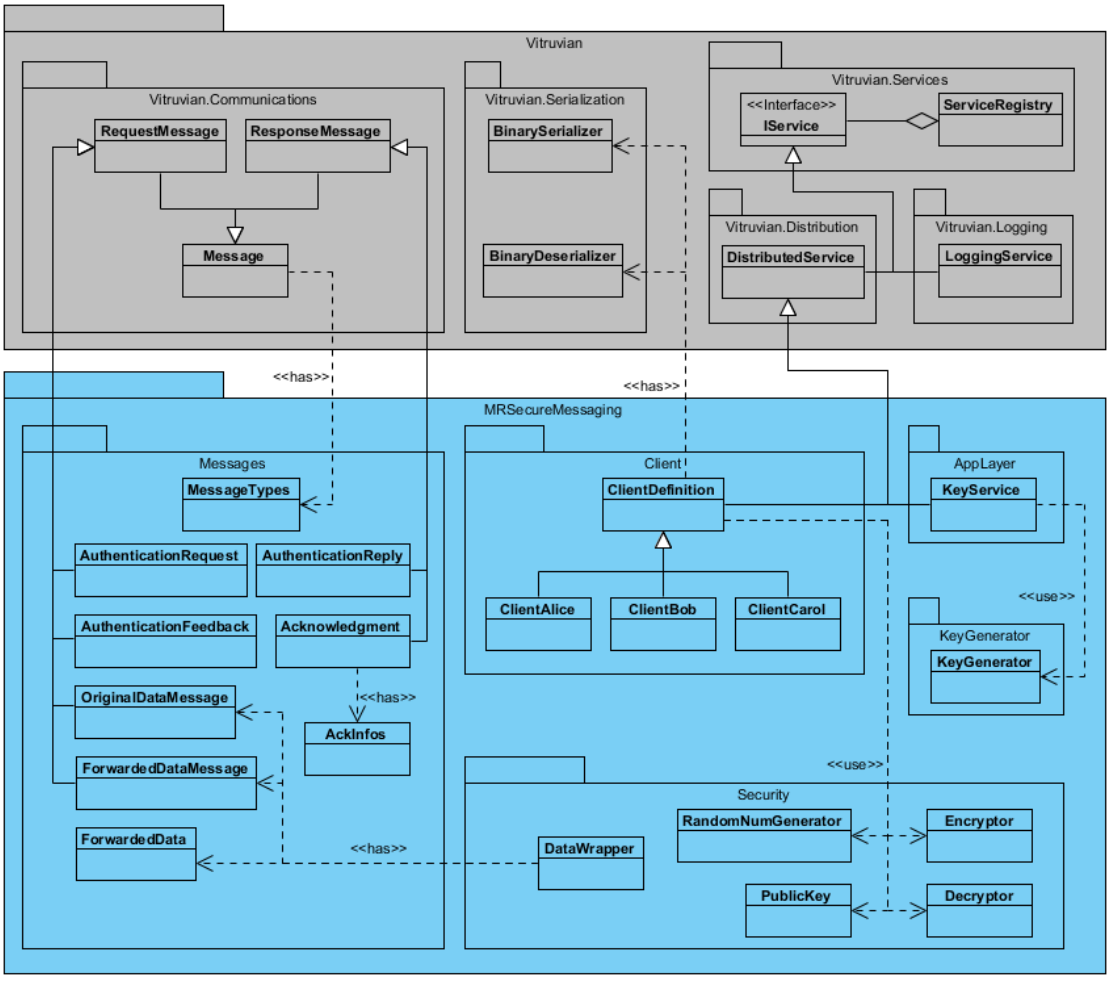


Figure 5-1: The MRSecureMessaging's abstract class diagram

Figure 5-1 shows the MRSecureMessaging's abstract class diagram. The package Vitruvian contains classes and methods for distributing and logging. It also contains classes and methods for serialization and communications. Specifically, the *KeyService* class implements an *IService* interface. In another word, it extends *DistributedService* class which implements an *IService* interface. This class contains the method required for distributing public keys. Package *Client* uses *BinarySerializer* and *BinaryDeserializer* that are defined in package *Vitruvian.Serialization*. Package *Messages* contains various types of messages which extend generic *Message* class in package *Vitruvian.Communicatons*.

## 5.2 Design Decisions and Dependencies of Package *Messages*

Authentication is a core functionality of the MRSecureMessaging. To realize this functionality, each principal needs various types of messages. Package *Messages* is designed for this purpose. Fortunately, Vitruvian has provided related classes in it. Figure 5-2 shows them in details.

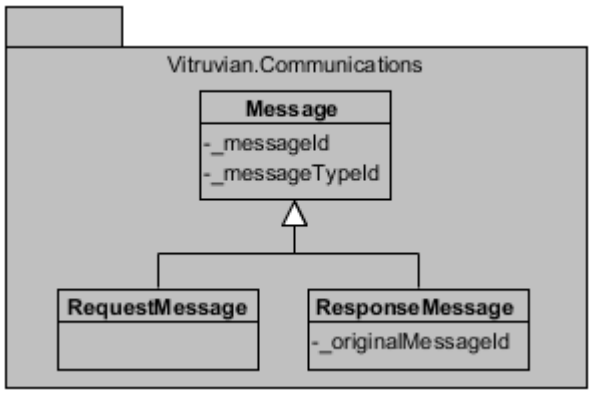


Figure 5-2: Message classes of Vitruvian

The *Messages* package contains four message classes that inherit from *RequestMessage* and two message classes inherit from *ResponseMessage*. *Acknowledge* is a reply to *AuthenticationFeedback*, *OriginalDataMessage* and *ForwardedDataMessage*. Obviously, it replies different requests with different information. *AckInfos* stores this information in it. Similarly, *MessageTypes* includes different message types (see Figure 5-3). *ForwardedData* contains data which needs to be forwarded in it.

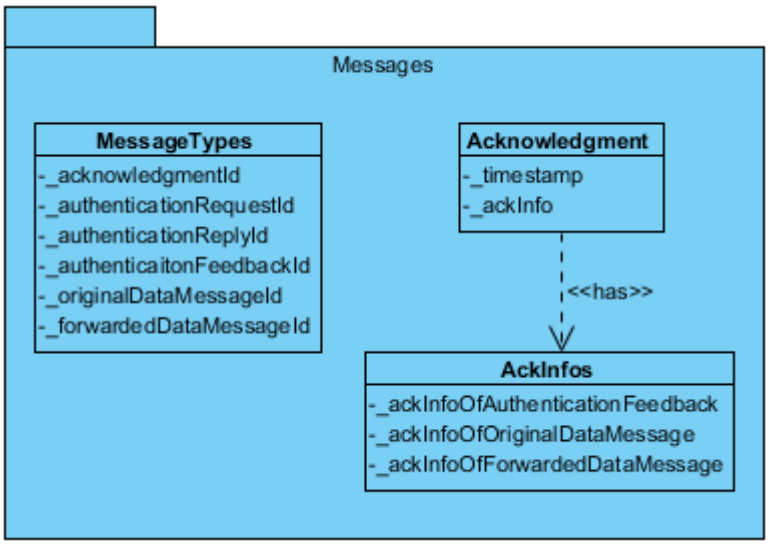


Figure 5-3: Messages classes of the MRSecureMessaging

## 5.3 Design Decisions and Dependencies of package *Security*

All security-related classes are defined in this package to improve reusability and reduce redundant code (see Figure 5-4).

As the names would imply, *Encryptor* and *Decryptor* are a pair of classes designed for encrypting and decrypting messages. They are designed in this way so

that reusability and flexibility are achieved. Each of them has a static method in it. When a client wants to encrypt/decrypt a message, it just needs to pass the corresponding key to *Encryptor*/*Decryptor*'s method. Since these two classes are in an independent package, they can also be used in other places as long as the cryptography algorithms are the same.

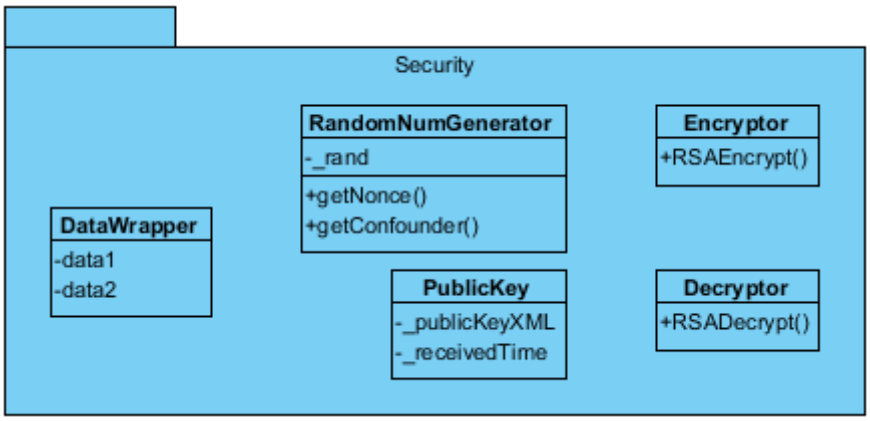


Figure 5-4: Security-related classes of the MRSecureMessaging

*RandomNumGenerator* generates nonces and confounders which both are random numbers essentially. The object of this class is also contained in *ClientDefinition* class.

*PublicKey* stores other principals' public key, but it also contains a timestamp in it to record the time when this public key is received. Once this public key expires, the client will ask *KeyService* for a new one.

*DataWrapper* is a simple wrapper which has two *byte* arrays in it to store two encrypted data. This class is closely related to three other classes: *ForwardedData*, *OriginaldataMessage* and *ForwardedDataMessage*. The first client would encrypt the

objects of the first two classes separately and put them into *DataWrapper* when he is sending data to the second client. The second client would remove the *OriginaldataMessage* in that wrapper object and substitute it with *ForwardedDataMessage* when he is forwarding data to the third client.

## 5.4 Design Decisions and Dependencies of package *Client*

This package contains classes and methods related to authentication protocol and data distribution. The Vitruvian framework is also used in it. *ClientDefinition* is the base class which contains all fundamental attributes in it. *ClientAlice*, *ClientBob* and *ClientCarol* inherit this class and contain authentication related methods respectively.

Clients need to communicate with each other. To realize this, *ClientDefinition* extends an abstract class called *DistributedService* in Vitruvian. This class simplifies the process of distributing applications, and makes it possible to distribute the application anywhere in the software development cycle. *BinarySerializer* and *BinaryDeserializer* are also defined in Vitruvian. *ClientDefinition* uses them to serialize messages or deserialize *byte* arrays. Other attributes like client name, client id, guest id and key storages are also contained in its definition (see Figure 5-5).

*ClientAlice*, *ClientBob* and *ClientCarol* have authentication related methods in them. These methods could be categorized into two types: local method and remote procedure call. Local method typically has three steps: first, generate a message; second, serialize this message into a *byte* array; third, encrypt this *byte* array with the

recipient's public key. Remote procedure call has six steps: first, decrypt the input with its secret key; second, deserialize the *byte* array from the first step into a message object; third, check if the information contained in this message is as expected or not. If yes, generate corresponding response message and follow the same three steps as used in local method. If no, log an error.

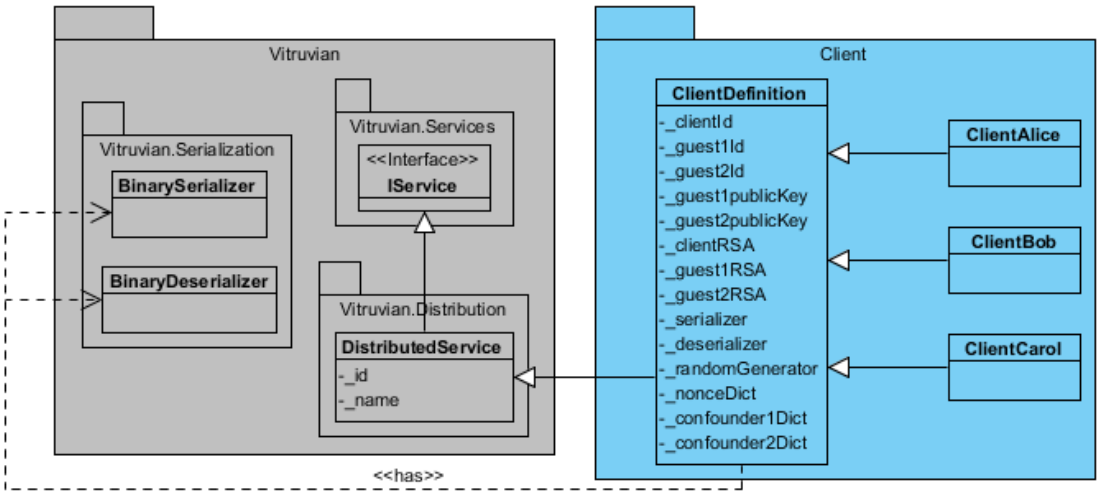


Figure 5-5: Client classes and their inheritance hierarchy

# CHAPTER 6

# IMPLEMENTATION DETAILS

The MRSecureMessaging is implemented in C# programming language on the .NET Framework 4.5, using the Vitruvian Framework to the handle key and data distribution. The following sections introduce what implementation challenges were met and how they were addressed. Section 6.1 gives a brief introduction to Vitruvian Framework and how it is used in the MRSecureMessaging. Section 6.2 describes the cryptography algorithm used in the MRSecureMessaging.

## 6.1 Introduction to Vitruvian Distribution

One of the key functionalities of the MRSecureMessaging is distributing various types of information (include public key, authentication messages and healthcare information). This is also the main challenge encountered when implementing it.

The Vitruvian Framework provides a service-oriented architecture that distributes objects using dynamically generated proxies that are specializations of the application classes [15]. These proxies use one or more *SyncPatterns* to manage the synchronization between all the replicas of a given patient object. Because the *SyncPattern* is external to the proxy, it can be changed dynamically. A proxy type overrides designated methods and properties in the original type and seamlessly

connects them to the synchronization patterns. A programmer can declare a synchronization pattern by simply adding an attribute to a property or method. Since the communication and synchronization are handled by the framework, there is no differentiation between a local object and a distributed object from a programmer's perspective.

The Vitruvian Framework has several different types of *SyncPatterns*. As described in Chapter 5, the *SyncPattern* used in the MRSecureMessaging is *Remote Procedure Call* or *RPC*. In RPC, procedures on remote machines can be called as if they are procedures in the local address space. The underlying RPC system then hides important aspects of distribution, including the encoding and decoding of parameters and results, the passing of messages and the preserving of the required semantics for the procedure call [16].

Take public key distribution as an example to illustrate this implementation details. First, at *KeyService* side, the function needs to be annotated like this:

```
[SyncPattern("RPC")]
virtual public string GetPublicKey(UInt16 clientId)
```

Figure 6-1: Sample remote procedure call code

A proxy of *KeyService* would be generated at client side. When a client is requesting a public key, it simply calls this proxy's *GetPublicKey* method as if it is a local method. Figure 6-2 shows the details of the remote proxy and Figure 6-3 shows the details of the local proxy ("r" means remote and "l" means local).

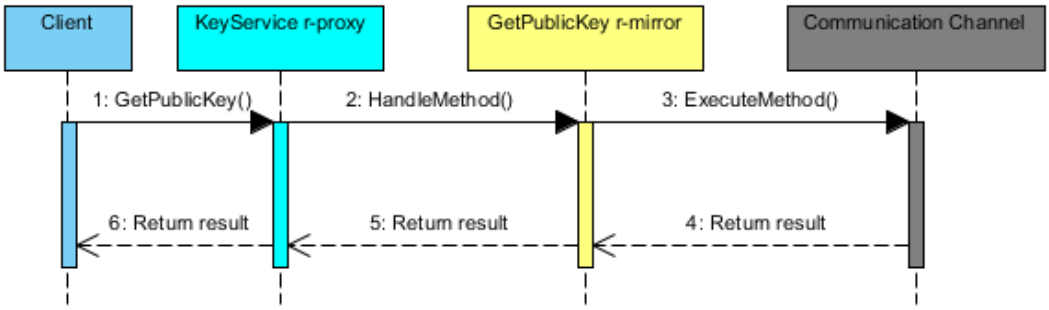


Figure 6-2: RPC remote proxy

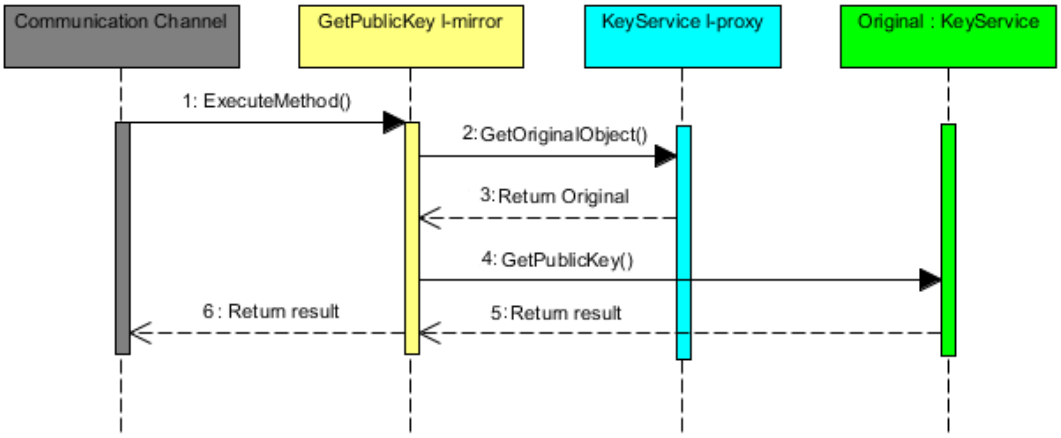


Figure 6-3: RPC local proxy

## 6.2 Cryptographic Algorithm

The MRSecureMessaging utilizes RSA algorithm [5] for encryption and decryption. It works as follow:

1. pick two large primes $p$ and $q$, let $n = p * q$
2. choose $e$ relatively prime to $\phi(n) = (p - 1)(q - 1)$
3. use Euclid's algorithm to generate a $d$ such that $e * d = 1 \bmod \phi(n)$
4. make the pair $(n, e)$ publicly available – this is the public key. The secret

key is $d$

5. a message block $M$ is now encrypted by calculating $C = M^e \ mod \ n$
6. the encrypted block $C$ is decrypted by calculating $M = C^d \ mod \ n$

The .Net Framework 4.5 provides the *RSACryptoServiceProvider* for this purpose. This class creates a public/secret key pair when the default constructor is used to create a new instance. After a new instance of the class is created, the key information can be exported in to two formats: either a string of XML representation or an *RSAParameters* structure. You can also use a Boolean value to indicate whether to return only the public key information or to return both the public key and the secret key information.

In *KeyGenerator*, keys are exported into XML representation. On *KeyService* side, only public keys are exported; On client side, both public keys and secret keys are exported.

Secret keys should never be stored verbatim or in plain text on the local computer. A key container is needed to store it. Specifically, *CspParameters* is used in the MRSecureMessaging. This is another class provided by the .Net Framework; it guarantees secret keys cannot easily be compromised.

# CHAPTER 7

# SOFTWARE TESTING

## 7.1 Introduction

Software testing is an investigation conducted to provide stakeholders with information about the quality of the product or service under test [31]. Software testing can also provide an objective, independent view of the software to allow the business to appreciate and understand the risks of software implementation [17]. Software testing involves the execution of a software component or system to evaluate one or more properties of interest. In general, these properties indicate the extent to which the component or system under test [17]:

- meets the requirements that guided its design and development,
- responds correctly to all kinds of inputs,
- performs its functions within an acceptable time,
- is sufficiently usable,
- can be installed and run in its intended environments, and
- achieves the general result its stakeholders desire.

From the above we could conclude that testing is more than just debugging. But debugging is still a very important part of testing. Software bugs will almost always exist in any software module with moderate size: not because programmers are

careless or irresponsible, but because the complexity of software is generally intractable, and humans have only limited ability to manager complexity. It is also true that for any complex systems, design defects can never be completely ruled out [18].

Testing is also expensive. Typically, more than 50% percent of the development time is spent in testing [18]. So testing is a trade-off between budget, time and quality.

The MRSecureMessaging is thoroughly tested. Section 6.2 discusses the unit testing and Section 6.3 explains the integration testing.

## 7.2 Unit Testing

Unite testing, also known as component testing, refers tests that verify the functionality of a specific section of code, usually at the function level. In an object-oriented environment, this is usually at the class level, and the minimal unit tests include the constructors and destructors [32]. Unit testing alone cannot verify the functionality of a piece of software, but rather is used to ensure that the building blocks of the software work independently from each other [17].

Extensive test cases have written for almost all classes in the MRSecureMessaging. These classes include various message classes, client classes, *KeyGenerator*, *KeyService*, *Encryptor* and *Decryptor*. Test cases were designed such that they met MRSecureMessaging's functional requirement discussed in Chapter 4. Different types of inputs that user may provide were also covered in those test cases.

## 7.3 Integration Testing

Integration testing is the phase in software testing in which individual software modules are combined and tested as a group. It occurs after unit testing. Integration testing takes as its input modules that have been unit tested, groups them in a larger aggregates, applies tests defined in an integration test plan to those aggregates, and delivers as its output the integrated system ready for further testing [19]. Integration testing works to expose defects in the interfaces and interaction between integrated components. Progressively larger groups of tested software components corresponding to elements of the architectural design are integrated and tested until the software works as a system [17].

For this project, testing focused on the flows of key/authentication/data from one component to another. Specifically, they are as follows:

1. Testing of communication between clients and *KeyService*
2. Testing of authentication process between clients
3. Testing of data (PII and other confidential data) flow between *ClientAlice* and *ClientBob*
4. Testing of data (coordinated, de-identified data) flow between ClientBob and ClientCarol

# CHAPTER 8

# SUMMARY

The MRSecureMessaging utilizes a reliable public key cryptographic algorithm to encrypt PII and other confidential data to make sure that phMPI and data consumers can only access their own data. It also contains the built-in customized Needham-Schroeder protocol that allows data sources to authenticate the intended recipients. Specifically, data sources first use it to verify recipients and then encrypt different types of data with different keys so that only corresponding recipients can decrypt it. This approach guarantees privacy, security and communication efficiency.

Although the MRSecureMessaging is specific to the ECCS, it can actually be used in any system where secure communication is needed. With minor changes, it could be used in two-client communication or multiple-client (more than three) communication. It can also be used in systems where access control is needed. The design of the MRSecureMessaging strictly follows the software engineering principles which allow high cohesion and loose coupling among classes. Modules are easy to understand, maintain and reuse. Changes in one module do not affect other modules.

Future work could focus on changing the cryptographic algorithms used in it. The MRSecureMessaging is using public key cryptographic algorithm right now. If

symmetric cryptographic algorithm is needed in future, as mentioned in Chapter 3, the communication architecture will be changed dramatically and becomes more complicated. The trade-off between these two architectures will need further research.

While working on this project, I had the opportunity to practice the theory and principles learned from software engineering class and object-oriented software development class. I also gained experience from almost all phases of the software development process. These precious experiences improved my skills in documenting, implementing and testing. It also improved my system design ability and coding style.

# REFERENCES

[1] Clark, John Andrew, and Jeremy Lawrence Jacob. A survey of authentication protocol literature Version 1.0. (1997).

[2] en.wikipedia.org/wiki/Symmetric-key_algorithm

[3] en.wikipedia.org/wiki/Public-key_encryption

[4] Diffie, Whitfield, and Martin Hellman. "New directions in cryptography." Information Theory, IEEE Transactions on 22.6 (1976): 644-654.

[5] R. Rivest, A. Shamir, and L. Adleman. A Method for Obtaining Digital Signatures and Public Key Cryptosystems. Communications of the ACM, 21(2):120–126, February 1978.

[6] Needham, Roger M., and Michael D. Schroeder. "Using encryption for authentication in large networks of computers." Communications of the ACM 21.12 (1978): 993-999.

[7] Federal Information Processing Standard 46 – the Data Encryption Standard, 1976.

[8] Gong, Lomas, et al. "Protecting poorly chosen secrets from guessing attacks." Selected Areas in Communications, IEEE Journal on 11.5 (1993): 648-656.

[9] Lowe, Gavin. "Breaking and fixing the Needham-Schroeder public-key protocol using FDR." Tools and Algorithms for the Construction and Analysis of Systems.

Springer Berlin Heidelberg, 1996. 147-166.

[10] en.wikipedia.org/wiki/Systems_analysis

[11] Eriksson, Hans-Eric; Penker, Magnus; Lyons, Brian; Fado, David. (2004). UML 2 Toolkit. Indianapolis, Indiana: Wiley Publishing Inc

[12] en.wikipedia.org/wiki/Functional_requirement

[13] en.wikipedia.org/wiki/Non-functional_requirement

[14] msdn.microsoft.com/en-us/library/ee658098.aspx

[15] B. Smith and S. Clyde, “An Orthogonal Approach To Distribution: An Introduction to the Vitruvian Framework”, 2008 9th IEEE/ACM International Conference on Grid Computing, pp. 192-200

[16] Coulouris G, Dollimore J, Kindberg T, Blair G. Distributed systems: concepts and design Fifth Edition. Massachusetts: Addison Wesley, 2012

[17] en.wikipedia.org/wiki/Software_testing

[18] users.ece.cum.edu/~koopman/des_s99/sw_testing/

[19] en.wikipedia.org/wiki/Integration_testing

[20] en.wikipedia.org/wiki/Secure_communication

[21] en.wikipedia.org/wiki/Man-in-the-middle_attack

[22] Smith, Brian G., "Two Highly Diverse Studies In Computing: A Vitruvian Framework For Distribution And A Search Approach To Cancer Therapies" (2008).

[23] Pakalapati, Aditya, "A Flexible Consent Management System for Master Person Indices" (2012).

[24] Dale G. O'Brien, William A. Yasnoff (1999). Privacy, confidentiality, and security in information systems of state health agencies: American Journal of Preventive Medicine, Volume 16, Issue 4

[25] McCallister, Erika. Guide to protecting the confidentiality of personally identifiable information. Diane Publishing, 2010.

[26] Centers for Disease Control and Prevention (CDC). "Data security and confidentiality guidelines for HIV, viral hepatitis, sexually transmitted disease, and tuberculosis programs: standards to facilitate sharing and use of surveillance data for public health action." Atlanta, GA: US Department of Health and Human Services, Centers for Disease Control and Prevention (2011).

[27] Goldreich, Oded. Foundations of Cryptography: Volume 2, Basic Applications. Vol. 2. Cambridge university press, 2004.

[28] en.wikipedia.org/wiki/Encryption

[29] en.wikipedia.org/wiki/Cryptographic_protocol

[30] en.wikipedia.org/wiki/Authentication_protocol

[31] Kaner, Cem (November 17, 2006). "Exploratory Testing". Florida Institute of Technology, Quality Assurance Institute Worldwide Annual Software Testing Conference, Orlando, FL. Retrieved November 22, 2014.

[32] Binder, Robert V. (1999). Testing Object-Oriented Systems: Objects, Patterns, and Tools. Addison-Wesley Professional. p. 45. ISBN 0-201-80938-9.